AD A017176

Stanford Artificial Intelligence Laboratory
Memo AIM-264

August 1975

Computer Science Department
Report No. STAN-CS-75-506, AIM-264

OPERATIONAL REASONING and DENOTATIONAL SEMANTICS

by

Michael Gordon

35 p.

DAHC15-73-C-0435, ARPA Order-2494

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

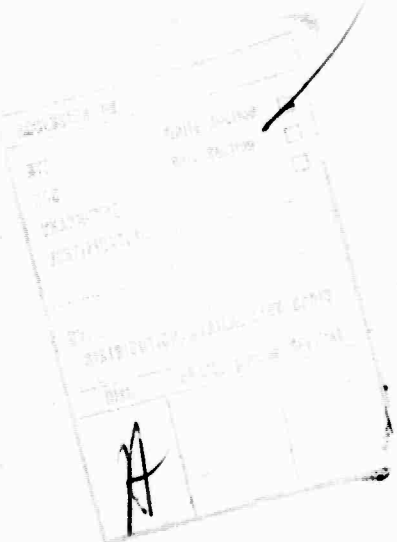
DDC
RECEIVED
NOV 14 1975
B

COMPUTER SCIENCE DEPARTMENT
Stanford University

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

094 120



OPERATIONAL REASONING AND DENOTATIONAL SEMANTICS

by

Michael Gordon
Department of Computer Science,
James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh EH9 3JZ.

Abstract

"Obviously true" properties of programs can be hard to prove when meanings are specified with a denotational semantics. One cause of this is that such a semantics usually abstracts away from the running process - thus properties which are obvious when one thinks about this lose the basis of their obviousness in the absence of it. To enable process-based intuitions to be used in constructing proofs one can associate with the semantics an abstract interpreter so that reasoning about the semantics can be done by reasoning about computations on the interpreter. This technique is used to prove several facts about a semantics of pure LISP. First a denotational semantics and an abstract interpreter are described. Then it is shown that the denotation of any LISP form is correctly computed by the interpreter. This is used to justify an inference rule - called "LISP-induction" - which formalises induction on the size of computations on the interpreter. Finally LISP-induction is used to prove a number of results. In particular it is shown that the function eval is correct relative to the semantics - i.e. that it denotes a mapping which maps forms (coded as S-expressions) on to their correct values.

ACKNOWLEDGEMENTS

Thanks to John Allen, Rod Burstall, Friedrich von Henke, Robert Milne, Gordon Plotkin, Bob Tennent and Chris Wadsworth for helpful discussions and correspondence. John Allen, Dana Scott and Akinori Yonezawa suggested improvements and pointed out errors in preliminary drafts of this report

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract DAHC 15-73-C-0435, ARPA order no. 2494.

The views and conclusions in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the US Government.

CONTENTS

1. Introduction

This paper contains examples of the use of operational reasoning to prove properties of a denotational semantics. By "operational reasoning" is meant reasoning which exploits notions associated with the operations involved in running programs on interpreters. "Obviously true" properties are often rather hard to prove when meanings are specified by a denotational semantics. One cause of this is that such a semantics usually abstracts away from the running process - thus properties which are obvious when one thinks about this lose the basis of their obviousness in the absence of it. One way to enable process-based intuitions to be used in constructing proofs is to associate with such a semantics an abstract interpreter so that one can reason about the semantics by reasoning about computations on the interpeter. In what follows this approach is used to prove several facts about a semantics of pure LISP. Doing this involves:

(A) Describing a set of semantic equations for pure LISP

(B) Describing an interpreter (expressed as a calculus) for mechanically evaluating LISP forms.

Having done this I then prove that the denotation of a form (as specified by the semantic equations) is always correctly computed by the interpreter. This result is then used to formulate a special purpose induction rule for reasoning about LISP programs. This rule - called "LISP-induction" - is induction on the length of computations on the interpreter. Because the interpreter is correct LISP-induction is valid for reasoning from the semantic equations. Using LISP-induction I outline how to prove the correctness of the LISP function eval. This involves showing that the denotation of eval (as specified by the semantic equations) is a mapping which maps LISP forms (coded as S-expressions) on to their correct values.

2. Syntax of Pure LISP

The syntax of LISP described below is that of M-expressions as described in the manual [4].

I use the variant of BNF notation described in [9]

2.1. Meta-variable Conventions

A	ranges	over	<S-expression>	(as in page 9 of [4])
x,f,z	range	over	<identifier>	(as in page 9 of [4])
e	ranges	over	<form>	(as defined below)
fn	ranges	over	<function>	(as defined below)
F	ranges	over	<standard function>	(as defined below)

I use meta-variables x,f,z to range over <identifier>; x is used in contexts where the identifier is a form, f where it's a function and z where it could be either.

2.2. BNF Equations

$$e ::= A \mid x \mid fn[e_1;...;e_n] \mid [e_{11} \rightarrow e_{12};...;e_{n1} \rightarrow e_{n2}]$$
$$fn ::= F \mid f \mid \lambda[[x_1;...;x_n];e] \mid label[f;fn] \mid \mu[f;fn]$$
$$F ::= car \mid cdr \mid cons \mid atom \mid eq$$

(The purpose and meaning of functions of the form $\mu[f;fn]$ is explained in [Note 16] below)

3. Denotational Semantics of Pure LISP

The formal definition of LISP described in this section is a "mathematical semantics" of the type developed by Scott and Strachey [9]. I have found that modelling data-types as complete lattices [8] leads to minor technical difficulties and inelegancies [1] which disappear if complete-posets (ie. partially ordered sets in which every directed set has a least upper bound) are used. Consequently in what follows - and in contrast to the standard theory - "domain" will mean "complete-poset". The theory based on this notion of domain differs only in obvious and trivial ways from the theory based on complete-lattices.

If X is a set let flat(X) be the domain obtained from X by adjoining $\perp$ to it and imposing the ordering $\perp \sqsubseteq x$ for all $x \in X$. Thus

$$x \in \text{flat}(X) \iff x = \perp \text{ or } x \in X$$
$$x \sqsubseteq y \iff x = \perp \text{ or } x = y$$

The following syntactic domains will be used later:

$$S = \text{flat}(\langle \text{S-expression} \rangle)$$
$$Id = \text{flat}(\langle \text{identifier} \rangle)$$
$$Form = \text{flat}(\langle \text{form} \rangle)$$
$$Function = \text{flat}(\langle \text{function} \rangle)$$

The semantics below should be read in conjunction with the explanatory notes that follow it.

3.1. Semantics

3.1.1. Denotation Domains

$$D = S + Funval \qquad \text{[Note 1]}$$
$$S = \text{flat}(\langle \text{S-expression} \rangle)$$
$$Funval = /S^* \rightarrow S/ \qquad \text{[Note 2]}$$

3.1.2. Environment Domain

$$Env = Id \rightarrow /Env \rightarrow D/ \qquad \text{[Note 3]}$$

3.1.3. Semantic Functions

$$\mathfrak{E}: Form \rightarrow /Env \rightarrow S/ \qquad \text{[Note 4]}$$
$$\mathfrak{F}: Function \rightarrow /Env \rightarrow Funval/ \qquad \text{[Note 4]}$$

3.1.4. Semantic Equations

(S1) $$\mathfrak{E}[\![A]\!]\rho = A \qquad \text{[Note 5]}$$

(S2) $$\mathfrak{E}[\![x]\!]\rho = \rho(x)\rho|S \qquad \text{[Note 6]}$$

(S3) $$\mathfrak{E}[\![fn[e_1;\ldots;e_n]]\!]\rho = \mathfrak{F}[\![fn]\!]\rho(\mathfrak{E}[\![e_1]\!]\rho,\ldots,\mathfrak{E}[\![e_n]\!]\rho) \qquad \text{[Note 7]}$$

(S4) $$\mathfrak{E}[\![[e_{11} \rightarrow e_{12}; \ldots; e_{n1} \rightarrow e_{n2}]]\!]\rho = (\mathfrak{E}[\![e_{11}]\!]\rho \rightarrow \mathfrak{E}[\![e_{12}]\!]\rho, \ldots, \mathfrak{E}[\![e_{n1}]\!]\rho \rightarrow \mathfrak{E}[\![e_{n2}]\!]\rho) \qquad \text{[Note 8]}$$

(S5)
$$\mathfrak{F}[\![car]\!]\rho = \underline{\lambda} s{:}S.\underline{car}(s) \qquad \text{[Note 9]}$$
$$\mathfrak{F}[\![cdr]\!]\rho = \underline{\lambda} s{:}S.\underline{cdr}(s) \qquad \text{[Note 10]}$$
$$\mathfrak{F}[\![cons]\!]\rho = \underline{\lambda} s_1, s_2{:}S.\underline{cons}(s_1, s_2) \qquad \text{[Note 11]}$$
$$\mathfrak{F}[\![atom]\!]\rho = \underline{\lambda} s{:}S.\underline{atom}(s) \qquad \text{[Note 12]}$$
$$\mathfrak{F}[\![eq]\!]\rho = \underline{\lambda} s_1, s_2{:}S.\underline{eq}(s_1, s_2) \qquad \text{[Note 13]}$$

(S6) $$\mathfrak{F}[\![f]\!]\rho = \rho(f)\rho|Funval \qquad \text{[Note 14]}$$

(S7) $$\mathfrak{F}[\![\lambda[[x_1;\ldots x_n];e]]\!]\rho = \underline{\lambda} s_1, \ldots, s_n{:}S.\mathfrak{E}[\![e]\!]\rho[s_1/x_1]\ldots[s_n/x_n] \qquad \text{[Note 15]}$$

(S8) $$\mathfrak{F}[\![label[f;fn]]\!]\rho = \mathfrak{F}[\![fn]\!]\rho[\mathfrak{F}[\![fn]\!]/f] \qquad \text{[note 16]}$$

(S9) $$\mathfrak{F}[\![\mu[f;fn]]\!]\rho = Y(\lambda v{:}/Env \rightarrow Funval/.\lambda \rho'{:}Env.\mathfrak{F}[\![fn]\!]\rho'[v/f])\rho \qquad \text{[Note 16]}$$

3.2. Notes

Note 1

"+" is the separated sum [6]. If D_1, D_2 are domains then:

$$D_1 + D_2 = \{(1,d_1) | d_1 \in D_1\} \cup \{(2,d_2) | d_2 \in D_2\} \cup \{\bot\}$$

which is made into a domain by imposing the ordering:

$$\bot \sqsubseteq (n,d)$$
$$(n,d) \sqsubseteq (n',d') \iff n=n' \text{ and } d \sqsubseteq d'$$

If $d_i \in D_i$ let $(d_i \text{ in} D)$ mean (i,d_i) - i.e. the natural injection of d_i into $D_1 + D_2$

If $d \in D_1 + D_2$ let

$$d|D_i = d_i \text{ , if } d=(i,d_i);$$
$$= \bot \text{ , otherwise.}$$

$d|D_i$ is the natural projection of d on to D_i.

Note 2

If L is a domain then:

$$L^* = \{(x_1,\ldots,x_m) \mid m \geq 0 \text{ and } x_i \in L\} \cup \{\bot\}$$

L^* is made into a domain by imposing the ordering:

$$\bot \sqsubseteq (x_1,\ldots,x_n)$$
$$(x_1,\ldots,x_n) \sqsubseteq (y_1,\ldots,y_m) \iff n=m \text{ and } \forall i.\ x_i \sqsubseteq y_i$$

Note 3

The purpose of environments in the semantics is like the purpose of alists in interpreters. Thus environments are used to hold the bindings of variables to their values. In LISP when a function is bound to a name on the alist the values of the function's free variables are not determined. These values depend on the environment in which the function is activated and this is unknown at definition time. To model this the objects which get bound to names on environments are mappings defined on environments. These objects thus have type $/Env \to D/$ and so Env has to have the circular type $Id \to /Env \to D/$. This kind of environment also handles the binding of form variables to S-expressions - the binding of A to x in ρ is represented by arranging that $\rho(x)\rho' = A$ for all $\rho' \in Env$. Bob Tennent has suggested [private communication] that this somewhat unnatural representation of form variable bindings could be avoided by letting Env have type given by $Env = Id \to /S + /Env \to Funval//$.

The domain equation $Env = Id \to /Env \to D/$ may have many solutions. These solutions can be ordered by regarding them as retracts (and hence members) of a universal space. Then the Env intended here is that represented by the retract $Y(\lambda e.Id \to /e \to D/)$ - see [1] or [8] for further explanation. This minimality is needed in the proof of the Main Theorem (see below).

Note 4

Relative to an environment $\rho \in Env$ the semantic functions $\mathfrak{E},\mathfrak{F}$ map forms e, and functions fn, onto their denotations $\mathfrak{E}[\![e]\!]\rho \in S$, $\mathfrak{F}[\![fn]\!]\rho \in Funval$ respectively. The semantic equations consist of a recursive, syntax-directed definition of $\mathfrak{E}$ and $\mathfrak{F}$. $\mathfrak{E}[\![\bot]\!] = \bot$ and $\mathfrak{F}[\![\bot]\!] = \bot$. The "emphatic" brackets $[\![,]\!]$ are a device due to Scott to increase the legibility of complex expressions - $[\![...]\!]$ always enclose pieces of LISP code.

Note 5

S-expressions denote themselves in all environments.

Note 6

"$\rho(x)\rho|S$" means the projection into S of $\rho(x)\rho\in D=S+Funval$. Since S-expressions are constant (i.e. their meaning is environment independent) they get represented by constant functions in $/Env\to S/$ (see [note 3]). Thus for any $\rho'\in Env$ "$\rho(x)\rho'$" would do just as well as the right hand side of (S2). However when f is a function name "$\rho(f)\rho$" is needed (see (S6) and [note 3]) and so it seems more elegant to have (S2) as it is so that it resembles (S6). As mentioned in [note 3] this arbitrariness is eliminated if the type of Env is changed to satisfy $Env=Id\to/S+/Env\to Funval//$.

Note 7

If $s_1,\ldots,s_n\in S$ then $(s_1,\ldots,s_n)\in S^*$ and for $f\in/S^*\to S/$ $f(s_1,\ldots,s_n)$ means $f((s_1,\ldots,s_n))$

Note 8

$(s_{11}\to s_{12},\ldots,s_{n1}\to s_{n2})$ = if $s_{11}=\perp$ or ($s_{11}\neq T$ and $s_{11}\neq F$) then $\perp$ else if $s_{11}=T$ then s_{12} else
If $s_{21}=\perp$ or ($s_{21}\neq T$ and $s_{21}\neq F$) then $\perp$ else if $s_{21}=T$ then s_{22} else
.
.
.
if $s_{n1}=\perp$ or ($s_{n1}\neq T$ and $s_{n1}\neq F$) then $\perp$ else if $s_{n1}=T$ then s_{n2} else $\perp$.

Note 9

See [note 15] for explanation of $\underline{\lambda}$.

$\underline{car}:S\to S$ is defined by: $\underline{car}(s) = A_1$, if $s=(A_1.A_2)$;
$= \perp$, otherwise (i.e. if $s=\perp$ or s is atomic).

Note 10

See [note 15] for explanation of $\underline{\lambda}$.

$\underline{cdr}:S\to S$ is defined by: $\underline{cdr}(s) = A_2$, if $s=(A_1.A_2)$;
$= \perp$, otherwise (i.e. if $s=\perp$ or s is atomic).

Note 11

See [note 15] for explanation of $\underline{\lambda}$.

$\underline{cons}:S\times S\to S$ is defined by: $\underline{cons}(s_1,s_2) = (s_1.s_2)$, if $s_1\neq\perp$ and $s_2\neq\perp$;
$= \perp$, otherwise.

Note 12

See [note 15] for explanation of _

$\underline{atom}:S\to S$ is defined by: $\underline{atom}(s) = T$, if s is atomic;
$= F$, if s is of the form $(s_1.s_2)$;
$= \perp$, otherwise (i.e. if $s=\perp$).

Note 13

See [note 15] for explantion of $\underline{\lambda}$.

$\underline{eq}:S\times S\to S$ is defined by: $\underline{eq}(s_1,s_2) = T$, if s_1 and s_2 are atomic and $s_1=s_2$;
$= F$, if s_1 and s_2 are atomic and $s_1\neq s_2$;
$= \perp$, otherwise.

Note 14

$\rho(f)\rho|Funval$ is the projection into *Funval* of $\rho(f)\rho\in D=S\to Funval$. Forming $\rho(f)$ models looking up f on the alist, applying $\rho(f)$ to ρ to get $\rho(f)\rho$ models looking up the free variables in the activation environment.

Note 15

Suppose $\mathbf{E}(s_1,...,s_n)$ is an expression which takes values in the domain D_2 when $s_1,...,s_n$ range over domain D_1 then $\underline{\lambda}s_1,...,s_n{:}D_1.\mathbf{E}(s_1,...,s_n)$ denotes the function $D_1{}^*\rightarrow D_2$ defined by

$f(s) = \mathbf{E}(s_1,...,s_n)$, if $s=(s_1...s_m)$ where $m\geq n$ and $\forall i.\ s_i\neq\perp$;
$= \perp$, otherwise.

Thus $\underline{\lambda}s_1,...,s_n{:}D_1.\mathbf{E}(s_1,...,s_n)$ is a function which always returns $\perp$ when one of its arguments is $\perp$ (this is to model call-by-value) and which can take any number of arguments $\geq n$ (this is a property of LISP functions).

I shall use $\lambda s_1,...,s_n{:}D_1.\mathbf{E}(s_1,...,s_n)$ (i.e. with λ instead of $\underline{\lambda}$) in the usual way to mean the function $f{:}D_1{}^n\rightarrow D_2$ defined by

$f(s_1,...,s_n) = \mathbf{E}(s_1,...,s_n)$

If $\rho\in Env$, $v\in /Env\rightarrow D/$ and $z\in Id$ then

$\rho[v/z] = \lambda z'{:}Id.$ If $z=\perp$ or $z'=\perp$ then $\perp$ else if $z=z'$ then v else $\rho(z')$

In (S7) I've used the "coercion conventions" that if $s\in S$ then $\rho[s/z]$ means $\rho[(\lambda\rho'{:}Env.(s\ in D))/z]$. Thus, as discussed in [Note 3], for the purposes of binding to variables in environments S-expressions are represented as constant functions.

Note 16

There are two natural ways to analyse recursion. One of them is to mimic in the semantic equations what the LISP eval function does - viz bind the function to it's own name on the alist. The other way is to take the denotation of a recursively defined function to be the (minimal) solution of the equation which defines it. Both these approaches have something to be said for them and fortunately they turn out to be equivalent in practice (what "in practice" means will be elaborated later - it's also discussed abstractly in [3]). To simplify the investigation of this equivalence two kinds of recursive functions, **label[f;fn]** and μ[f;fn], are included in the syntax of LISP. **label**[f;fn] is given an analysis which mimics the eval function whilst μ[f;fn] receives a minimal-fixed-point treatment.

In both (S8) and (S9) I've used the "coercion convention" that if $v \in /Env \to Funval/$ then $\rho[v/f]$ is to mean $\rho[(\lambda\rho' : Env.(v(\rho') \text{ in } D))/f]$.

In (S9) $Y : //Env \to Funval/ \to /Env \to Funval// \to /Env \to Funval/$ is the usual minimal-fixed-point operator $\lambda F.\bigsqcup_n F^n(\bot)$. Note that the fixed point extraction is done "before" the free variables are looked up (i.e. the result of applying Y is applied to ρ, rather than Y being applied to something which has already been applied to ρ). This is necessary to correctly model dynamic binding (fluid variables).

Note also that from the fixed point property of Y we have:

$$\mathfrak{F}[\![\mu[f;fn]]\!]\rho = \mathfrak{F}[\![fn]\!](\rho[\mathfrak{F}[\![\mu[f;fn]]\!]/f])$$

The right hand side of this differs subtly from that of (S8).

4. An Interpreter for Pure LISP

The interpreter described below is designed so that reasoning about computations on it is convenient. Its purpose is to aid in the formulation of a special purpose induction rule for LISP ("LISP-induction"). It is formalised as a calculus consisting of rules for simplifying terms of the form $\langle e \mid a \rangle$ where e is a LISP form and a an alist. The rules of this calculus are intended to correspond to the obvious simplifications that one would perform on expressions of the form $\mathcal{G}[\![e]\!]\rho$. An example of such a simplification is:

$$\begin{aligned}
&\mathcal{G}[\![\lambda[[x];[atom[x]\rightarrow x;T\rightarrow cdr[x]]]((1\ 2)/\bot)]\!]\\
&\quad=\mathcal{G}[\![[atom[x]\rightarrow x;T\rightarrow cdr[x]]]\!](\bot[(1\ 2)/x])\\
&\quad=\mathcal{G}[\![cdr[x]]\!](\bot[(1\ 2)/x])\\
&\quad=\mathcal{G}[\![cdr[(1\ 2)]]\!](\bot[(1\ 2)/x])\\
&\quad=(2)
\end{aligned}$$

Let the meta-variables p,a range over the strings defined by:

p::=A | $\langle$e | a$\rangle$
a::=NIL | a[A/z] | a[fn/z]

p will be said to range over <term> and a over <alist>. Define $\mathfrak{B}[\![p]\!]\in S$ and $\mathfrak{A}[\![a]\!]\in Env$ by

$$\begin{aligned}
\mathfrak{B}[\![A]\!]&=A\\
\mathfrak{B}[\![\langle e \mid a\rangle]\!]&=\mathcal{G}[\![e]\!](\mathfrak{A}[\![a]\!])
\end{aligned}$$

$$\begin{aligned}
\mathfrak{A}[\![NIL]\!]&=\bot\\
\mathfrak{A}[\![a[A/z]]\!]&=\mathfrak{A}[\![a]\!][A/z]\\
\mathfrak{A}[\![a[fn/z]]\!]&=\mathfrak{A}[\![a]\!][\mathfrak{F}[\![fn]\!]/z]
\end{aligned}$$

In the last two equations I've used the "coercion conventions" described in [note 15] and [note 16] above.

The following definition describes a binary relation $\longrightarrow$ defined on terms. $p\longrightarrow p'$ means that p simplifies to p'. If one likes one can think of the p's as states of a machine then "$p\longrightarrow p'$" means "in state p move to state p'", final states are p's of the form A.

I shall immediately follow the definition of $\longrightarrow$ with an explanation of the notation it is written in; then I will give some notes which should be read in conjunction with the definition.

Definition 1 (Definition of $\longrightarrow$ and $\overset{*}{\longrightarrow}$)

$\overset{*}{\longrightarrow}$ is the reflexive, transitive closure of $\longrightarrow$.

(P1) $\langle A \mid a\rangle \longrightarrow A$

(P2) $$\frac{a(x)=A}{\langle x \mid a\rangle \longrightarrow A}$$ [note 17]

(P3) $$\frac{\underline{F}(A_1,\ldots,A_n)=A}{\langle F[A_1;\ldots;A_n] \mid a\rangle \longrightarrow A}$$ [note 18]

(P4) $$\frac{[\forall i.\ \langle e_i \mid a\rangle \overset{*}{\longrightarrow} A_i] \text{ and } [\exists i.\ e_i \neq A_i]}{\langle fn[e_1;\ldots;e_n] \mid a\rangle \longrightarrow \langle fn[A_1;\ldots;A_n] \mid a\rangle}$$ [note 19]

(P5) $$\frac{\langle e_{m1} \mid a\rangle \overset{*}{\longrightarrow} T \text{ and } [\forall i<m.\ \langle e_{i1} \mid a\rangle \overset{*}{\longrightarrow} F]}{\langle [e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}] \mid a\rangle \longrightarrow \langle e_{m2} \mid a\rangle}$$

(P6) $$\frac{a(f)=fn}{\langle f[A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle fn[A_1;\ldots;A_n] \mid a\rangle}$$ [note 20]

(P7) $$\frac{m\leq n}{\langle \lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle e \mid a[A_1/x_1]\ldots[A_m/x_m]\rangle}$$

(P8) $\langle label[f;fn][A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle fn[A_1;\ldots;A_n] \mid a[fn/f]\rangle$

(P9) $\langle \mu[f;fn][A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle fn[A_1;\ldots;A_n] \mid a[\mu[f;fn]/f]\rangle$

(P10) $$\frac{[n=1] \text{ or } [1\leq n \text{ and } p_1 \longrightarrow p_2,\ldots,p_{n-1} \longrightarrow p_n]}{p_1 \overset{*}{\longrightarrow} p_n}$$ [note 21]

Each clause P1-P10 is a schema and the meta-variables in them range over their previously defined sets (e.g. A ranges over <S-expression>).

A schema of the form $p \longrightarrow p'$ (i.e.P1, P8 or P9) means that any instance of it is a pair for which $\longrightarrow$ holds.

A schema of the form:

$$\frac{\text{conditions}}{p \longrightarrow p'}$$

(i.e. P2-P7, P10) means that any instance of it which satisfies the conditions is a pair for which $\longrightarrow$ holds.

An example computation, which corresponds to the simplifications described above, is:

$\langle \lambda[[x];[atom[x] \rightarrow x; T \rightarrow cdr[x]]][(1\ 2)] \mid NIL \rangle$
$\longrightarrow \langle [atom[x] \rightarrow x; T \rightarrow cdr[x]] \mid NIL[(1\ 2)/x] \rangle$ (by P7)
$\longrightarrow \langle cdr[x] \mid NIL[(1\ 2)/x] \rangle$ (by P5)
$\longrightarrow \langle cdr[(1\ 2)] \mid NIL[(1\ 2)/x] \rangle$ (by P4)
$\longrightarrow (2)$ (by P3)

Notice that this computation can be mechanically and deterministically generated from its initial term - the definition of $\longrightarrow$ makes explicit the intuitions which were previously used in simplifying $\mathfrak{G}[\![\lambda[[x];[atom[x] \rightarrow x; T \rightarrow cdr[x]]][(1\ 2)]]\!](\perp)$ above.

4.1. Notes

Note 17

a(x) is defined by structural induction on a as follows:

$$\begin{aligned} NIL(x) &= \bot \\ (a[A/z])(x) &= \text{if } x=z \text{ then } A \text{ else } a(x) \\ (a[fn/z])(x) &= \text{if } x=z \text{ then } fn \text{ else } a(x) \end{aligned}$$

Thus $a(x) \in \{\bot\} \cup$ <S-expression> $\cup$ <function>. The reason P2 rather than "$\langle x \mid a\rangle \longrightarrow a(x)$" is used is that with the latter if $a(x)=\bot$ or $a(x)=fn$ then $\langle x \mid a\rangle \longrightarrow \bot$ or $\langle x \mid a\rangle \longrightarrow fn$ and neither $\bot$ nor fn are terms.

Note 18

F ranges over car,cdr,cons,atom,eq. "$\langle F[A_1;\ldots;A_n] \mid a\rangle \longrightarrow F(A_1,\ldots,A_n)$" won't do because it would yield e.g. $\langle cons[NIL] \mid a\rangle \longrightarrow \bot$ - and $\bot$ isn't a term.

Note 19

The reason for the condition "$[\exists i.\ e_i \neq A_i]$" in P4 is to exclude unending computations of the form:

$$\langle fn[A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle fn[A_1;\ldots;A_n] \mid a\rangle \longrightarrow \ldots$$

and also to make $\longrightarrow$ deterministic (i.e $p \longrightarrow p'$ and $p \longrightarrow p''$ => $p'=p''$). Thus I exclude the nondeterminism:

$$\langle label[f;fn][A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle fn[A_1;\ldots;A_n] \mid a[fn/f]\rangle$$
$$\langle label[f;fn][A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle label[f;fn][A_1;\ldots;A_n] \mid a\rangle$$

Note 20

a(f) is defined as in [note 17] (with x replaced by f in the definition of a(x)). "$\langle f[A_1;\ldots;A_n] \mid a\rangle \longrightarrow \langle a(f)[A_1;\ldots;A_n] \mid a\rangle$" will not do for P6 because of the possibility that $a(f)=\bot$ or $a(f)=A$ (c.f. [note 17]).

Note 21

It follows from P10 that $\forall p.\ p \twoheadrightarrow p$ (take n=1 in P10) and $p \twoheadrightarrow p'$ and $p' \twoheadrightarrow p'' \Rightarrow p \twoheadrightarrow p''$

5. Correctness of the Interpreter

The following result shows that $\longrightarrow$ fulfils its design requirements i.e. that $\langle e \mid a\rangle$ simplifies down to A if and only if $\mathfrak{F}[\![e]\!](\mathfrak{A}[\![a]\!])=A$

Theorem 1 (Main Theorem)

$p \twoheadrightarrow A \iff \mathfrak{B}[\![p]\!]=A$

Proof outline

The theorem splits into two parts, viz:

(a) $p \twoheadrightarrow A \Rightarrow \mathfrak{B}[\![p]\!]=A$
(b) $\mathfrak{B}[\![p]\!]=A \Rightarrow p \twoheadrightarrow A$

(a) is essentially trivial - one just checks that rules (P1)-(P10) preserve the denotation of terms. I discuss how to organize this argument in section 6.1. below.

(b) is less straightforward and I shall only indicate the main idea of the proof. This idea is due to Robert Milne [private communication] and considerably shortens the original proof given in [1]. Similar ideas have been developed independently by Reynolds[7].

The main idea is to construct predicates $\mathfrak{P}_i^{form}$, $\mathfrak{P}_i^{function}$, and $\mathfrak{P}_i^{alist}$ defined on $/Env \to S/ \times$<form>, $/Env \to Funval/ \times$<function> and $Env \times$<alist> respectively such that:

(1) $\mathfrak{P}_i^{form}(v,e) \iff \forall \rho,a.\ [\ \mathfrak{P}_i^{alist}(\rho,a) \Rightarrow \forall A.\ [\ v(\rho)=A \Rightarrow \langle e \mid a\rangle \twoheadrightarrow A\ ]\ ]$

(2) $\mathfrak{P}_i^{function}(v,fn) \iff \forall \rho,a.\ [\ \mathfrak{P}_i^{alist}(\rho,a) \Rightarrow [\ v(\rho)$ is strict] and $\forall A,A_1,\ldots,A_n.$
$[v(\rho)(A_1,\ldots,A_n)=A \Rightarrow \langle fn[A_1;\ldots;A_n] \mid a\rangle \twoheadrightarrow A\ ]\ ]$

(3) $\mathfrak{P}_i^{alist}(\rho,a) \iff \forall z,A.\ [$ if $a(z)=A$ then $\mathfrak{P}_i^{form}(\rho(z)|/Env \to S/,A)$ and
if $a(z)=fn$ then $\mathfrak{P}_i^{function}(\rho(z)|/Env \to Funval/,fn)\ ]$

In (3) above "$\rho(z)|/Env \to S/$" and "$\rho(z)|/Env \to Funval/$" are abbreviations for "$\lambda\rho' : Env.(\rho(z)\rho'|S)$" and "$\lambda\rho' : Env.(\rho(z)\rho'|$Funval)" respectively.

From (1)-(3) it is straightforward to show by structural induction that:

(4) $\forall e \in$<form>. $\mathfrak{P}_i^{form}(\mathfrak{E}[\![e]\!],e)$

(5) $\forall fn \in$<function>. $\mathfrak{P}_i^{function}(\mathfrak{F}[\![fn]\!],fn)$

(6) $\forall a \in$<alist>. $\mathfrak{P}_i^{alist}(\mathfrak{A}[\![a]\!],a)$

and then by taking $v=\mathfrak{E}[\![e]\!]$ and $\rho=\mathfrak{A}[\![a]\!]$ we have by (1), (6) and modus ponens that:

$\mathfrak{E}[\![e]\!](\mathfrak{A}[\![a]\!])=A \Rightarrow \langle e \mid a\rangle \twoheadrightarrow A$

as desired.

The only non trivial part ot this proof is showing that there exist relations $\mathfrak{P}_i^{form}$, $\mathfrak{P}_i^{function}$,$\mathfrak{P}_i^{elict}$ satisfying (1)-(3). Lack of monotonicity prohibits the simple use of Y to do this. General techniques for solving recursive predicate equations (such as (1)-(3) above) have been developed by Robert Milne (and also by Reynolds). The reader is referred to [5] and [7] for further details.

"QED"

5.1. Reasoning via the Interpreter

I'll start by illustrating the use of the Main Theorem on a totally trivial example - determining $\mathfrak{F}[\![label[f;f]]\!](\perp)$ - less trivial examples are theorems 2, 3, 4, 5 below. Intuitively $\mathfrak{F}[\![label[f;f]]\!](\perp)=\perp$ as label[f;f] terminates on no arguments - to rigorise this observe that by P8 we have for arbitrary $A_1,...,A_n$

$$\langle label[f;f][A_1;...;A_n] \mid NIL\rangle \longrightarrow \langle f[A_1;...;A_n] \mid NIL[f/f]\rangle$$

and (by P6) if $p=\langle f[A_1;...;A_n] \mid NIL[f/f]\rangle$ then the evaluation of p just leads to the unending computation:

$$p \longrightarrow p \longrightarrow p \longrightarrow ...$$

so by the Main Theorem there's no A such that $\mathfrak{G}[\![label[f;f][A_1;...;A_n]]\!](\perp)=A$ and so :

$$\forall A_1,...,A_n.\ \mathfrak{G}[\![label[f;f][A_1;...;A_n]]\!](\mathfrak{A}[\![NIL]\!])=\mathfrak{F}[\![label[f;f]]\!](\perp)(A_1,...,A_n)=\perp$$

hence $\mathfrak{F}[\![label[f;f]]\!](\perp)=\perp$

To prove the intuitively obvious fact that $\mathfrak{F}[\![label[f;f]]\!](\perp)=\perp$ without using the Main Theorem one needs to exploit the minimality of *Env*. The Main Theorem packages-up this minimality in an easy to use form.

6. LISP-Induction

LISP-induction is an attempt to formalize certain kinds of intuitive arguments about LISP programs. A very simple example of such an argument is the reasoning used at the end of the last section to prove that $\mathfrak{F}[\![\text{label}[f;f]]\!](\mathfrak{A}[\![a]\!])=\perp$. A less trivial example is the "proof" that

$$\forall a,f,fn.\ \mathfrak{F}[\![\text{label}[f;fn]]\!](\mathfrak{A}[\![a]\!])=\mathfrak{F}[\![\mu[f;fn]]\!](\mathfrak{A}[\![a]\!])$$

which is based on the intuition that for all $A_1,\ldots,A_n$ if one starts "evaluating" both sides of the equation

$$\mathfrak{F}[\![\text{label}[f;fn]]\!](\mathfrak{A}[\![a]\!])(A_1,\ldots,A_n)=\mathfrak{F}[\![\mu[f;fn]]\!](\mathfrak{A}[\![a]\!])(A_1,\ldots,A_n)$$

then either both "evaluations" will stop with the same value or both will go on for ever.

To convert this argument into reliable proof one needs a formal notion of evaluation (which has the property that unending evaluations only arise from terms which denote $\perp$). The definition of $\longrightarrow$ is designed to provide such a notion and the Main Theorem shows that it has the desired property.

Using $\longrightarrow$ one can give a more rigorous version of the above "proof" by showing that to any computation of the form

$$\langle \text{label}[f;fn][A_1;\ldots;A_n] \mid a\rangle \longrightarrow p_1 \longrightarrow p_2 \longrightarrow \ldots \longrightarrow p_n \longrightarrow A$$

there corresponds one of the form

$$\langle \mu[f;fn][A_1;\ldots;A_n] \mid a\rangle \longrightarrow p_1' \longrightarrow p_2' \longrightarrow \ldots \longrightarrow p_n' \longrightarrow A$$

and vice versa, where (roughly!) p_i' is got from p_i by replacing some occurrences of label by μ and replacing some alist bindings of the form [fn/f] by $[\mu[f;fn]/fn]$.

The LISP-induction rule to be described provides a reasonably clean way of rigorously organising such arguments. In order to state it let $p' \lhd^* p$ mean intuitively "p' has to be evaluated in the course of evaluating p ". More precisely let $\lhd^*$ be the transitive closure of $\lhd$ where:

$p' \lhd p \iff$ either (1) $p \longrightarrow p'$

or (2) $p = \langle fn[e_1;\ldots;e_n] \mid a\rangle$ and $p' = \langle e_i \mid a\rangle$ for some i.

or (3) $p = \langle [e_{11} \rightarrow e_{12}; \ldots; e_{n1} \rightarrow e_{n2}] \mid a\rangle$ and $p' \in \{\langle e_{11} \mid a\rangle, \ldots, \langle e_{m1} \mid a\rangle\}$ where $\langle e_{m1} \mid a\rangle \twoheadrightarrow T$ and $\forall i < m.\ \langle e_{i1} \mid a\rangle \twoheadrightarrow F$

Thus $p' \lhd^* p \iff \exists p_1,\ldots,p_n.\ p' = p_1 \lhd p_2 \lhd \ldots \lhd p_n = p \ (n>1)$

LISP-induction is structural (or Noetherian) induction with respect ot the ordering $\lhd^*$ applied to expressions of the form "$p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)$" where $\mathfrak{R}(p,A)$ is some sentence involving p and a. Thus the rule is:

$$\frac{\forall p.\ [\ [\forall p' \lhd^* p.\ [p' \twoheadrightarrow A' \Rightarrow \mathfrak{R}(p',A')]] \Rightarrow [p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)]\ ]}{\forall p.\ [p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)]}$$

By considering the various ways in which we can have $p' \lhd^* p$ the above rule can be instantiated to:

6.1. Simple LISP-Induction

TO INFER: $\forall p.\ p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)$

PROVE:

(1) $\mathfrak{R}(A,A)$

(2) $\mathfrak{R}(\langle A \mid a\rangle,A)$

(3) $a(x)=A \Rightarrow \mathfrak{R}(\langle x \mid a\rangle,A)$

(4) $\underline{F}(A_1;\ldots;A_n) \Rightarrow \mathfrak{R}(\langle F[A_1;\ldots;A_n] \mid a\rangle,A)$

(5) $\mathfrak{R}(\langle e_i \mid A_i\rangle,A_i)$, $\mathfrak{R}(\langle fn[A_1;\ldots;A_n] \mid a\rangle,A) \Rightarrow \mathfrak{R}(\langle fn[e_1;\ldots;e_n] \mid a\rangle,A)$

(6) $\forall i<m.\mathfrak{R}(\langle e_i \mid a\rangle,F)$, $\mathfrak{R}(\langle e_{m1} \mid a\rangle,T)$, $\mathfrak{R}(\langle e_{m2} \mid a\rangle,A) \Rightarrow \mathfrak{R}(\langle [e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}] \mid a\rangle,A)$

(7) $\mathfrak{R}(\langle e \mid a[A_1/x_1]\ldots[A_n/x_n]\rangle,A) \Rightarrow \mathfrak{R}(\langle \lambda[[x_1;\ldots;x_n];e][A_1;\ldots;A_n] \mid a\rangle,A)$

(8) $\mathfrak{R}(\langle fn[A_1;\ldots;A_n] \mid a[fn/f]\rangle,A) \Rightarrow \mathfrak{R}(\langle label[f;fn][A_1;\ldots;A_n] \mid a\rangle,A)$

(9) $\mathfrak{R}(\langle fn[A_1;\ldots A_n] \mid a[\mu[f;fn]/f]\rangle,A) \Rightarrow \mathfrak{R}(\langle \mu[f;fn][A_1;\ldots;A_n] \mid a\rangle,A)$

The above instance is somewhat less general than full LISP-induction and so it's called simple LISP-induction. Simple LISP-induction, however, is powerful enough to be used to to prove a number of interesting facts, for example here's the easy half of the Main Theorem:

Theorem 2

$p \twoheadrightarrow A \Rightarrow \mathfrak{B}[\![p]\!]=A$

Proof

Take $\mathfrak{R}$ to be such that $\mathfrak{R}(p,A) \Leftrightarrow \mathfrak{B}[\![p]\!]=A$, then the result follows from a trivial application of simple LISP-induction.

QED.

Theorem 3

$\mathfrak{F}[\![\mu[f;fn]]\!](\mathfrak{A}[\![a]\!]) = \mathfrak{F}[\![label[f;fn]]\!](\mathfrak{A}[\![a]\!])$

Proof

For the induction to go through one needs to prove a stronger result. If w, w' are forms or functions let w~w' if and only if w' can be got from w by changing zero or more occurrences of μ to label and zero more occurrences of label to μ.

If a,a' are alists define a~a' <=> for all z:

(1) a(z)=A <=> a'(z)=A (A∈<S-expression>)

(2) if a(z)=fn then [a'(z)=fn' or a'(z)=μ[f;fn']] where fn~fn'

(3) if a'(z)=fn' then [a(z)=fn or a(z)=μ[f;fn]] where fn~fn'.

If p,p' are terms then p~p' <=> [p=A=p' or p=$\langle$e | a$\rangle$, p'=$\langle$e' | a'$\rangle$ where e~e',a~a'].

Now one can use simple LISP-induction to verify that $p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)$ where.

$\mathfrak{R}(p,A)$ <=> [$\forall p'.\ p\sim p' \Rightarrow p' \twoheadrightarrow A$]

The result follows.

QED.

The previous theorem can't be generalised to:

$\forall \rho,f,fn.\ \mathfrak{F}[\![\mu[f;fn]]\!]\rho = \mathfrak{F}[\![label[f;fn]]\!]\rho$

A counterexample is got by taking fn=g, $\rho = Y(\lambda\rho''.\perp[\lambda\rho'.\rho'(f)\rho''/g][\mathfrak{F}[\![car]\!]/f])$. It is then straightforward to show (see [1] or [3]) that

$\mathfrak{F}[\![\mu[f;g]]\!]\rho = \perp \neq \mathfrak{F}[\![car]\!] = \mathfrak{F}[\![label[f;g]]\!]\rho$

Thus it's not the case that $\mathfrak{F}[\![\mu[f;fn]]\!]=\mathfrak{F}[\![label[f;fn]]\!]$. A detailed and LISP-independent discussion is given in [3].

Because variables are fluid in LISP it isn't true that if ρ,ρ' agree on the free variables of fn ther $\mathfrak{F}[\![fn]\!]\rho=\mathfrak{F}[\![fn]\!]\rho'$ (e.g. consider fn=, $\rho=\mathfrak{A}[\![NIL[g/f][car/g]]\!]$, $\rho'=\mathfrak{A}[\![NIL[g/f][cdr/g]]\!]$). The following definition gives sufficient conditions on a set $Z\subseteq$<Identifier> so that if a,a' agree on Z then $\mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a]\!])=\mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a']\!])$.

Definition 2

If $Z\subseteq$<Identifier> and p,p' are terms then define $p=_Z p'$ <=>

either p=p'=A

or p=⟨e | a⟩, p'=⟨e | a'⟩ and (1),(2) and (3) where:
(1) Z contains all the free variables in e
(a variable is free if it isn't bound by λ,μ or label)
(2) $\forall z\in Z.\ a(z)=a'(z)$
(3) $\forall z\in Z.$ Z contains all the free variable in a(z)

Theorem 4

$p=_Z p' \Rightarrow [\ p \twoheadrightarrow A \Leftrightarrow p' \twoheadrightarrow A\ ]$

Proof

Use simple LISP-induction to show that $p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)$ where:

$\mathfrak{R}(p,A) \Leftrightarrow \forall p'.\ [\ [\exists Z.\ p=_Z p'] \Rightarrow p' \twoheadrightarrow A\ ]$

The result then follows from the symetry of $=_Z$.

QED

Corollary

Let fn∈<function>, a, a'∈<alist> then if there is a Z⊆<identifier> such that:

(1) Z contains the free variables of fn

(2) ∀z∈Z. Z contains the free variables of a(z)

(3) ∀z∈Z. a(z)=a'(z)

Then $\mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a]\!])=\mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a']\!])$

Proof

By previous theorem $\langle fn[A_1;...;A_n] \mid a\rangle \rightarrow A \iff \langle fn[A_1;...;A_n] \mid a'\rangle \rightarrow A$ hence result by Main Theorem.

QED.

Results similar to Theorem 4 and its corollary are proved in a more general setting in [3].

7. The correctness of eval and apply

The properties of eval and apply which constitute their correctness are:

$$\mathfrak{E}[\![eval[e^*;a^*]]\!](\rho_{int})=\mathfrak{E}[\![e]\!](\mathfrak{A}[\![a]\!])$$
$$\forall A_1,...,A_n.\ \mathfrak{E}[\![apply[fn^*;(A_1...A_n);a^*]]\!](\rho_{int})=\mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a]\!])(A_1,...,A_n)$$

where e*,a* are S-expression representations of e and a and ρ_{int} is an environment binding the names of the various functions used in the definitions of eval and apply to their values (see below)

The proof to be outlined is not an instance of simple LISP-induction but is a general Noetherian induction with respect to the ordering $\prec$. The full details are very long and boring (see [1]) and are not given here - I hope that I describe enough so that it would be quicker for the reader to generate the proof himself than to read through it.

In fact the above properties are not true for if e=x (so $e^*=X$) and a=NIL[fn/x] (so $a^*=((X.fn^*))$) then

$$\mathfrak{E}[\![eval[e^*;a^*]]\!](\rho_{int})=fn^*\neq\perp=\mathfrak{E}[\![x]\!](\mathfrak{A}[\![a]\!])$$

However if we adhere to the constraint (violated above) that an identifier can't be used both as a form variable and a function name in the same program then the property holds.

To enable us to say this precisely we make the following definition:

Definition 3

$\langle e \mid a\rangle$ is "nice" if the intersection of the sets FORMVARS, FUNVARS are empty, where:

FORMVARS=$\{z|z$ is a form variable in e or $a(z)\in$<form>$\}$
FUNVARS=$\{z|z$ is a function name in e or $a(z)\in$<function>$\}$

The next definition extends the translation of M-expressions into S-expressions which is given in the Manual [4] to include alists. This is necessary for the statement of the correctness of eval and apply - viz. Theorem 5 below.

Definition 4 (definition of e*, fn*, a*)

The S-expression representation e*, fn*, a* of e, fn, a are defined by structural induction as follows:

e*:

$$A^* = (\text{QUOTE } A)$$
$$x^* = X$$
$$fn[e_1;\ldots;e_n]^* = (fn^*\ e_1^*\ldots e_n^*)$$
$$[e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}]^* = (\text{COND } (e_{11}^*\ e_{12}^*)\ldots(e_{n1}^*\ e_{n2}^*))$$

fn*:

$$car^* = CAR$$
$$cdr^* = CDR$$
$$cons^* = CONS$$
$$atom^* = ATOM$$
$$eq^* = EQ$$
$$\lambda[[x_1;\ldots;x_n];e]^* = (\text{LAMBDA } (x_1^*\ldots x_n^*)\ e^*)$$
$$label[f;fn]^* = (\text{LABEL } f^*\ fn^*)$$

a*:

$$NIL^* = NIL$$
$$a[A/z]^* = ((z^*.A).a^*)$$
$$a[fn/z]^* = ((z^*.fn^*).a^*)$$

a_{int}, which is specified in the next definition, is an alist containing the definitions of the functions which make up a basic LISP interpreter - namely those functions needed in defining eval and apply. The environment denoted by a_{int} is ρ_{int}.

Definition 5 (Specification of a_{int}, ρ_{int})

$\rho_{int}=\mathfrak{M}[\![a_{int}]\!]$

where:

a_{int}= NIL$[fn_{assoc}/assoc][fn_{pairlis}/pairlis][fn_{equal}/equal]$
$[fn_{null}/null][fn_{cadar}/cadar][fn_{caddr}/caddr][fn_{cadr}/cadr]$
$[fn_{cdar}/cdar][fn_{caar}/caar][fn_{evlis}/evlis][fn_{evcon}/evcon]$
$[fn_{eval}/eval][fn_{apply}/apply]$

where fn_{name} is the definition of name given in the manual [4].

for example:

```
fn_apply=λ[[fn;x;a];
          [atom[fn]→[eq[fn;CAR]→caar[x];
                    eq[fn;CDR]→cdar[x];
                    eq[fn;CONS]→cons[car[x];cadr[x]];
                    eq[fn;ATOM]→atom[car[x]];
                    eq[fn;EQ]→eq[car[x];cadr[x]];
                         T →apply[eval[fn;a];x;a]];
           eq[car[fn];LAMBDA]→eval[caddr[fn];pairlis[cadr[fn];x;a]];
           eq[car[fn];LABEL]→apply[caddr[fn];x;cons[cons[cadr[fn];caddr[fn]];a]]]]

fn_eval=λ[[e;a];
          [atom[e]→cdr[assoc[e;a]];
           atom[car[e]]→[eq[car[e];QUOTE]→cadr[e];
                        eq[car[e];COND]→evcon[cdr[e];a];
                             T    →apply[car[e];evlis[cdr[e];a];a]];
           T → apply[car[e];evlis[cdr[e];a];a]]]


fn_evcon=λ[[c;a];
          [eval[caar[c];a]→eval[cadar[c];a];
                T     →evcon[cdr[c];a]]]

fn_evlis=λ[[m;a];
          [null[m]→NIL;
             T   →cons[eval[car[m];a];evlis[cdr[m];a]]]]
```

Theorem 5 (correctness of eval, apply)

If $\langle e \mid a\rangle$ and $\langle fn[A_1;\ldots;A_n] \mid a\rangle$ are nice then:

$$\mathfrak{F}[\![eval[e^*;a^*]]\!](\rho_{int}) = \mathfrak{F}[\![e]\!](\mathfrak{A}[\![a]\!])$$
$$\mathfrak{F}[\![apply[fn^*;(A_1\ldots A_n);a^*]]\!](\rho_{int}) = \mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a]\!])(A_1,\ldots,A_n)$$

Proof

The theorem follows from lemma 1 and lemma 2 below.

QED.

Lemma 1

$$\langle fn[A_1;\ldots;A_n] \mid a\rangle \twoheadrightarrow A \;\Rightarrow\; \mathfrak{F}[\![apply[fn^*;(A_1\ldots A_n);a^*]]\!](\rho_{int}) = A$$
$$\langle e \mid a\rangle \twoheadrightarrow A \;\Rightarrow\; \mathfrak{F}[\![eval[e^*;a^*]]\!](\rho_{int}) = A$$

Proof

The lemma can be put into the form $p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)$ by defining

$$\mathfrak{R}(p,A) \iff \text{if } p=\langle fn[A_1;\ldots;A_n] \mid a\rangle \twoheadrightarrow A \text{ then } \mathfrak{F}[\![apply[fn^*;(A_1\ldots A_n);a^*]]\!](\rho_{int}) = A$$
$$\text{and if } p=\langle e \mid a\rangle \twoheadrightarrow A \text{ then } \mathfrak{F}[\![eval[e^*;a^*]]\!](\rho_{int}) = A$$

A straightforward (but tedious) LISP-induction then yields the lemma.

QED.

Lemma 2 below is a kind of generalised converse of lemma 1. The extra generality consists in proving the result for certain alists of the form $a_{int}[w_1/z_1]\ldots[w_n/z_n]$ instead of just for a_{int}. This extra generality is needed to enable the induction to go through.

The alists in question are those of the form $a_{int}.a'$ where a' is "safe" - here "$a_{int}.a'$" is defined by structural induction by:

$$a_{int}.NIL = a_{int}$$
$$a_{int}.(a[A/z]) = (a_{int}.a)[A/z]$$
$$a_{int}.(a[fn/z]) = (a_{int}.a)[fn/z]$$

Also an alist a is called "safe" if when

$Z=\{assoc,pairlis,equal,null,cadar,caddr,cadr,cdar,caar,evlis,evcon,eval,apply\}$

then: $\forall z \in Z.\ a(z)=\perp$.

These definitions imply that if Z is as above then for any safe a: $a_{int} =_Z (a_{int}.a)$. This fact needs to be used in the proof of lemma 2 below.

Lemma 2

If $\langle fn[A_1;...;A_n] \mid a\rangle$ and $\langle e \mid a\rangle$ are nice and a' is safe then:

$$\langle apply[fn^*;(A_1...A_n);a] \mid a_{int}.a'\rangle \twoheadrightarrow A \Rightarrow \mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a]\!])(A_1,...,A_n)=A$$
$$\langle eval[e^*;a^*] \mid a_{int}.a'\rangle \twoheadrightarrow A \Rightarrow \mathfrak{G}[\![e]\!](\mathfrak{A}[\![a]\!])=A$$

Proof

The lemma can be put in the form:

$$p \twoheadrightarrow A \Rightarrow \mathfrak{R}(p,A)$$

by defining

$$\mathfrak{R}(p,A) \iff \text{if } p=\langle apply[fn^*;(A_1...A_n);a^*] \mid a_{int}.a'\rangle \twoheadrightarrow A \text{ (where } a' \text{ is safe)}$$
$$\text{then } \mathfrak{F}[\![fn]\!](\mathfrak{A}[\![a]\!])(A_1,...,A_n)=A$$
$$\text{and if } p=\langle eval[e^*;a^*] \mid a_{int}.a'\rangle \twoheadrightarrow A \text{ (where } a' \text{ is safe)}$$
$$\text{then } \mathfrak{G}[\![e]\!](\mathfrak{A}[\![a]\!])=A$$

This can then be proved by a straightforward (but extremely tedious) LISP-induction.

QED.

8. Concluding Remarks

Although these proofs formalize intuitive arguments their size, when all details are filled in, is excessive. As these details are fairly mechanical and don't require creative acts for their generation a proof production system (such as FOL at Stanford or the new LCF at Edinburgh) should be able to help us cope with them. Another possibility is that abstract "high level" notions can be developed which encapsulate some of the facts (proved here for LISP) in a language independent form. A start at this has been attempted in [3]. Abstract notions help in the handling of large masses of detail by assisting in the isolation of those things which are language specific from those which are more universal. When the proofs of language independent facts are factored out from the proofs of the theorems described above the latter are made shorter and more direct (see [3]). The formulation of such high level, language independent notions should also assist in the design of proof construction systems - research into proof generation needs to proceed hand in hand with research into the structure of the proofs whose generation is desired.

8. References

[1] Gordon, M.J.C. (1973) **Models of pure LISP**. Experimental Programming Reports:No.31. Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh.

[2] Gordon, M.J.C. (1975) **Operational Reasoning and Denotational Semantics**. Presented at the International Symposium on Proving and Improving Programs, Arc-et-Senans, France (proceedings available from IRIA). Revised as Memo AIM 264, Computer Science Department, Stanford University.

[3] Gordon, M.J.C. (1975) **Towards a Semantic Theory of Dynamic Binding**. Memo AIM 265 , Computer Science Department, Stanford University.

[4] McCarthy, J. et.al. (1969) **LISP 1.5 Programmer's Manual**. MIT Press.

[5] Milne, R. (1974) **The formal semantics of computer languages and their implementations**. Oxford University Computing Laboratory, Programming Research Group, Technical Monograph PRG-13 (available on microfiche).

[6] Reynolds, J.C. (1972) **Notes on a Lattice-Theoretic Approach to the Theory of Computation**. Systems and Information Science, Syracuse University.

[7] Reynolds, J.C. (1974) **On the Relation between Direct and Continuation Semantics**. Second colloquium on Automata, Languages, and Programming. Saarbrucken.

[8] Scott, D. (1974) **Data Types as Lattices**. To appear as Springer Lecture Notes.

[9] Scott, D. and Strachey, C. (1972) **Towards a Mathematical Semantics for Computer Languages**. Proc. Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Vol.21, Polytechnic Institute of Brooklyn.